# Generalized fold expressions

Photo credit: Jo Nakashima

Jackie Kay

⚡ C++ Now 2017 ⚡

# C++17 fold expressions are great

```
template<bool... Pack>
constexpr auto and_() {
  return (... && Pack);
}
```

# Left fold with initial value

Credit: Bryce's presentation from Tuesday

```
template <typename... Ts>
void print(Ts&&... ts)
{
  (std::cout << ... << std::forward<Ts>(ts)) << "\n";
}
```

# With comma operator

```
template<typename F, typename... Ts>
void for_each(F&& f, Ts&&... ts) {
  ([&f, &ts]() {
    f(ts);
  }, ...);
}
```

# What about return values?

# Generalized fold one-liner

Credit: [Barry Revzin on Stack Overflow]

```
template <typename F, typename Z, typename... Xs>
auto fold_left(F&& f, Z acc, Xs&&... xs) {
  ((acc = f(acc, xs)), ...);
  return acc;
}
```

# Works with constexpr too

```
template <typename F, typename Z, typename... Xs>
constexpr auto fold_left(F&& f, Z acc, Xs&&... xs) {
  ((acc = f(acc, xs)), ...);
  return acc;
}

auto sum = [](auto x, auto y){ return x + y; };

static_assert(fold_left(sum, 1, 2, 3) == 6);
```

# Gotcha

Only works if return type of all intermediate operations is uniform. Consider this case:

```
auto concatenated_tuple = fold_left(
    [](auto&& x, auto&& y) {
      return std::tuple_cat(x, y);
    },
    std::make_tuple("hello world"),
    std::make_tuple(1, 2, 3),
    std::make_tuple(std::vector<float>{})
);
```

# Possible solution

```
template<typename F, typename X>
struct fold_wrapper {
  F f;
  X state;

  template<typename Arg>
  constexpr auto operator>>=(Arg&& arg) {
    auto result = f(state, arg.state);
    return fold_wrapper<F, decltype(result)>{f, result};
  }
};
template <typename F, typename... Xs>
constexpr auto fold_left(const F& f, Xs&&... xs) {
  auto result = (... >>= fold_wrapper<F, Xs>{f, xs});
  return result.state;
}
```